Go Advanced

A Complete 5-Chapter Course

Topics covered:

Generics: Type Parameters and Constraints

Error Wrapping: errors.Is, errors.As, and %w

context.Context

Testing: the testing Package and go test

JSON and encoding/json

Exercises: 15 hands-on challenges with sample solutions

Format: A4 · Dark-theme code examples · Quick-reference tables

Table of Contents

CHAPTER 1 OF 5

Generics: Type Parameters and Constraints

COURSE 3 · CH 1

Generics: Type Parameters and Constraints

Writing one function or type that works correctly across many concrete types, without giving up Go's compile-time type safety

Chapter 5 (Fundamentals) showed that `sum(numbers ...int)` only works for `int` — a separate function would be needed for `float64`, despite the logic being identical. Generics, added to Go relatively recently, solve exactly this: one function definition, usable with multiple types, fully checked at compile time. Unlike Intermediate Chapter 2's interfaces (which work via shared methods), generics work via shared *capability* — "any type that supports +", for example.

The Problem Without Generics

```
func sumInts(numbers []int) int {
        total := 0
        for _, n := range numbers {
                total += n
        }
        return total
}

func sumFloats(numbers []float64) float64 {
        total := 0.0
        for _, n := range numbers {
                total += n
        }
        return total
}
```

These two functions are identical except for one type. Before generics, this duplication was simply accepted, or worked around with `any` (Intermediate Chapter 2) at the cost of losing type safety and needing manual type assertions.

A Generic Function with a Type Parameter

```
func Sum[T int | float64](numbers []T) T {
        var total T
        for _, n := range numbers {
                total += n
        }
        return total
}

fmt.Println(Sum([]int{1, 2, 3}))         // 6
fmt.Println(Sum([]float64{1.5, 2.5}))    // 4
```

`[T int | float64]` declares `T` as a **type parameter**, constrained to either `int` or `float64` — `T` stands in for whichever concrete type is actually used at the call site. `var total T` uses Chapter 2's zero-value behaviour generically: it's `0` when `T` is `int`, `0.0` when `T` is `float64`, decided automatically. Go infers `T` from the argument — there's no need to write `Sum[int](...)` explicitly in normal use.

Named Constraints

```
type Number interface {
        int | float64
}

func Sum[T Number](numbers []T) T {
        var total T
        for _, n := range numbers {
                total += n
        }
        return total
}
```

Inline constraints like `int | float64` get repetitive across several generic functions, so they're usually pulled out into a named **constraint interface** instead — an interface listing allowed underlying types rather than methods, reusable by name across the whole package.

> **COMPARABLE — A BUILT-IN CONSTRAINT**
>
> Go provides a built-in constraint called `comparable`, satisfied by any type that supports `==` and `!=` — commonly used for generic functions that need to check for a specific value inside a collection, covered in this chapter's challenges.

A Generic Type — Not Just a Generic Function

```go
type Stack[T any] struct {
        items []T
}

func (s *Stack[T]) Push(item T) {
        s.items = append(s.items, item)
}

func (s *Stack[T]) Pop() T {
        last := s.items[len(s.items)-1]
        s.items = s.items[:len(s.items)-1]
        return last
}

intStack := &Stack[int]{}
intStack.Push(10)
intStack.Push(20)
fmt.Println(intStack.Pop())   // 20
```

Structs (Intermediate Chapter 1) can be generic too: `Stack[T any]` declares a stack that works with whatever single type `T` is chosen at creation time — `Stack[int]{}` here. Every method automatically carries the same `T`, so `Push` and `Pop` stay perfectly type-safe for whichever type was chosen, with no `any`-style runtime checking required anywhere inside.

GENERICS ARE NOT THE SAME AS ANY

`any` (Intermediate Chapter 2) gives up type information entirely — the compiler can't help at all. Generics keep full compile-time type checking; `Stack[int]` and `Stack[string]` are still fully separate, type-safe usages, just generated from one shared definition.

APPROACH	TYPE SAFETY	CODE REUSE
Separate function per type	Full	None — duplicated logic
any / interface{}	None at compile time	Full, but unsafe
`Generics — func F[T Constraint](...)`	Full	Full

Coding Challenges

CHALLENGE 1

Write a generic function Max[T int | float64](a, b T) T that returns the larger of two values. Call it once with two ints and once with two float64s, printing both results.

View solution

CHALLENGE 2

Write a generic function Contains[T comparable](items []T, target T) bool that returns true if target appears anywhere in items. Test it once with a []string and once with a []int.

View solution

CHALLENGE 3

Using the Stack[T any] type from this chapter, create a Stack[string], push 3 names onto it, then pop and print all 3 (which should come back out in reverse order).

View solution

CHAPTER 1 QUICK REFERENCE (ADVANCED)

- **func F[T Constraint](x T) T { }** — a generic function with type parameter T
- **[T int | float64]** — inline constraint: T must be one of these listed types
- **type Name interface { typeA | typeB }** — a reusable, named constraint
- **comparable** — built-in constraint for types supporting == and !=
- **any** — the loosest constraint; equivalent to no constraint at all
- **type Name[T any] struct { }** — a generic type; methods carry the same T automatically
- **Generics ≠ any** — full compile-time type checking is preserved throughout
- **Next chapter:** error wrapping — errors.Is, errors.As, and fmt.Errorf("%w", err)

CHAPTER 2 OF 5

Error Wrapping: errors.Is, errors.As, and %w

COURSE 3 · CH 2

Error Wrapping: errors.Is, errors.As, and %w

Adding context to an error as it travels up through several function calls, without losing the ability to check what it originally was

Fundamentals Chapter 5 introduced `error` and `nil`. In real programs, an error often passes through several layers of function calls before reaching code that decides what to do about it — and each layer usually wants to add context ("failed while loading config" on top of "file not found"). **Error wrapping** is how Go adds that context without throwing away the original error underneath.

A Sentinel Error — A Known, Comparable Error Value

```
import "errors"

var ErrNotFound = errors.New("item not found")

func findItem(id int) (string, error) {
        if id != 1 {
                return "", ErrNotFound
        }
        return "Widget", nil
}
```

A **sentinel error** — a specific, named `error` value declared once at package level — lets calling code check for that exact error later. This is the foundation everything else in this chapter builds on: a known error value that can be compared against.

Wrapping an Error with fmt.Errorf and %w

```go
func loadItem(id int) (string, error) {
        name, err := findItem(id)
        if err != nil {
                return "", fmt.Errorf("loadItem: %w", err)
        }
        return name, nil
}

_, err := loadItem(5)
fmt.Println(err)   // loadItem: item not found
```

`%w` (used only with `fmt.Errorf`) wraps the original error inside a new one, attaching extra context (`"loadItem: "`) while keeping the original — `ErrNotFound` — retrievable from inside the new error. This differs from `%v` or `%s`, which would only produce a plain string with no way back to the original error value.

errors.Is — Checking for a Specific Error Through Any Number of Wraps

```go
_, err := loadItem(5)

if errors.Is(err, ErrNotFound) {
        fmt.Println("That item genuinely doesn't exist.")
} else if err != nil {
        fmt.Println("Some other error occurred:", err)
}
// That item genuinely doesn't exist.
```

`err != ErrNotFound` (a direct comparison) would actually return `true` here, since `err` is now the wrapped version, not the original — direct comparison breaks once wrapping is involved. `errors.Is(err, ErrNotFound)` unwraps as many layers as necessary, checking whether `ErrNotFound` is anywhere inside the chain. This is the correct, wrap-aware way to ask "is this ultimately that specific error?"

> **NEVER COMPARE A WRAPPED ERROR DIRECTLY WITH ==**
>
> Once an error has passed through `fmt.Errorf("...: %w", err)` even once, it is a brand-new error value — `==` against the original sentinel will always be false. `errors.Is` exists specifically to make this comparison correctly regardless of how many wrapping layers exist.

Custom Error Types

```
type ValidationError struct {
        Field string
        Message string
}

func (e *ValidationError) Error() string {
        return fmt.Sprintf("%s: %s", e.Field, e.Message)
}

func validateAge(age int) error {
        if age < 0 {
                return &ValidationError{Field: "age", Message: "cannot be negative"}
        }
        return nil
}
```

Any type with an `Error() string` method automatically satisfies Go's built-in `error` interface (Intermediate Chapter 2's structural typing again, applied to a single specific method). A custom error type like `ValidationError` can carry structured data — `Field`, `Message` — rather than just a flat message string.

errors.As — Extracting a Custom Error Type

```
err := validateAge(-5)

var valErr *ValidationError
if errors.As(err, &valErr) {
        fmt.Println("Invalid field:", valErr.Field)
}
// Invalid field: age
```

`errors.As` checks whether an error (possibly wrapped) matches a specific custom error *type*, and if so, assigns it into `valErr` so its specific fields (`Field`, `Message`) become accessible — something a plain `error` interface value alone never allows. `errors.Is` answers "is this *that exact value*?"; `errors.As` answers "is this *that type*, and if so, give it back to me properly typed."

TOOL	QUESTION IT ANSWERS
`fmt.Errorf("...: %w", err)`	Add context while preserving the original error
`errors.Is(err, sentinel)`	"Is this ultimately THIS specific error value?"
`errors.As(err, &target)`	"Is this (or something it wraps) of THIS type? Give it to me."
`err == sentinel`	Unsafe once wrapping is involved — avoid

Coding Challenges

CHALLENGE 1

Declare var ErrEmptyName = errors.New("name cannot be empty"). Write a function greet(name string) error that returns this sentinel (wrapped with fmt.Errorf and extra context "greet: %w") if name is "". Use errors.Is to check the result and print an appropriate message.

View solution

CHALLENGE 2

Define a custom error type RangeError with fields Min and Max ints, and an Error() string method describing the valid range. Write a function checkAge(age int) error returning a *RangeError if age is outside 0-120. Use errors.As to extract it and print Min/Max.

View solution

CHALLENGE 3

Write three functions that call each other: layerOne calls layerTwo, layerTwo calls layerThree, and layerThree returns a sentinel error ErrFailed. Each layer wraps the error with its own name using fmt.Errorf("...: %w", err). Print the final error from layerOne, then confirm errors.Is(err, ErrFailed) is still true despite 3 layers of wrapping.

View solution

CHAPTER 2 QUICK REFERENCE

- **var ErrX = errors.New("...")** — a sentinel error, a known comparable value
- **fmt.Errorf("context: %w", err)** — wraps err, adding context, keeping it unwrappable
- **errors.Is(err, sentinel)** — true if sentinel appears anywhere in err's wrap chain
- **errors.As(err, &target)** — extracts a specific custom error TYPE from the chain
- **type X struct {} + func (e *X) Error() string** — defines a custom error type
- **Never use == on a wrapped error** — it will not match, even when logically "the same"
- **Next chapter:** context.Context — cancellation, timeouts, and request-scoped values

CHAPTER 3 OF 5

context.Context

COURSE 3 · CH 3

context.Context: Cancellation, Timeouts, and Request-Scoped Values

One value, passed through every layer of a call chain, that can say "stop now" to everything downstream at once

Intermediate Chapter 3 showed goroutines running independently, with no automatic way for `main()` to know when they're done short of a channel or `WaitGroup`. `context.Context` solves a related but different problem: how does code running several layers deep — goroutines, function calls, network requests — find out that it should **stop**, because a timeout passed or the caller gave up? There's no real JavaScript equivalent to reach for here; the closest conceptual cousin is an `AbortController`, but Go's version is used far more pervasively.

The Basic Pattern — Pass ctx as the First Parameter

```
import "context"

func doWork(ctx context.Context) {
        select {
        case <-ctx.Done():
                fmt.Println("Work cancelled:", ctx.Err())
        case <-time.After(2 * time.Second):
                fmt.Println("Work finished normally")
        }
}
```

By strong convention, `ctx context.Context` is always the **first** parameter of any function that supports cancellation — never buried among other arguments. `ctx.Done()` returns a channel (Intermediate Chapter 3) that something closes the moment cancellation happens; `select` here waits on whichever of two channels is ready first, a new construct specifically for working with multiple channels at once.

context.WithTimeout — Cancel Automatically After a Duration

```
ctx, cancel := context.WithTimeout(context.Background(), 1 * time.Second)
defer cancel()   // always defer cancel — releases resources even if it times out first

doWork(ctx)
// Work cancelled: context deadline exceeded
```

`context.Background()` creates a brand-new, empty starting context — the root every other context derives from. `context.WithTimeout(parent, duration)` wraps it with a deadline: after that duration, `ctx.Done()` closes automatically, and `ctx.Err()` reports `context deadline exceeded`. `doWork` here finishes via the timeout branch instead of its normal 2-second wait, since the 1-second deadline fires first.

> **ALWAYS DEFER CANCEL(), EVEN ON SUCCESS**
>
> `WithTimeout` (and its cousin `WithCancel`) both return a `cancel` function that must be called to release the context's internal resources, whether or not the timeout ever actually triggers. `defer cancel()` immediately after creating the context is the standard, safe habit — skipping it can leak resources in long-running programs.

context.WithCancel — Manual Cancellation

```
ctx, cancel := context.WithCancel(context.Background())

go func() {
        time.Sleep(500 * time.Millisecond)
        cancel()   // some other goroutine decides it's time to stop
}()

doWork(ctx)
// Work cancelled: context canceled
```

`WithCancel` hands back a `cancel` function with no automatic timer attached — cancellation happens only when something explicitly calls it, from anywhere that has a reference to that function. This is the pattern used when one part of a program needs to tell every dependent goroutine to stop, on demand, rather than after a fixed duration.

Passing Request-Scoped Values

```go
type contextKey string

const userIDKey contextKey = "userID"

ctx := context.WithValue(context.Background(), userIDKey, 42)

func handleRequest(ctx context.Context) {
        userID := ctx.Value(userIDKey)
        fmt.Println("Handling request for user", userID)
}
```

`context.WithValue` attaches a single key/value pair to a context, retrievable anywhere downstream via `ctx.Value(key)` — most commonly used for request-scoped data in a web server (a user ID, a trace ID) that many unrelated layers of code might need without explicitly threading it through every function signature.

DON'T USE CONTEXT.WITHVALUE FOR ORDINARY FUNCTION PARAMETERS

It's tempting to reach for `WithValue` as a way to avoid adding parameters to a function — resist this. It's intended specifically for cross-cutting, request-scoped data (tracing IDs, auth info), not as a general substitute for normal arguments, which stay clearer and more type-safe as actual parameters.

TOOL	PURPOSE
`context.Background()`	The root context everything else derives from
`context.WithTimeout(parent, d)`	Auto-cancels after duration d; always defer cancel()
`context.WithCancel(parent)`	Manual cancellation via the returned cancel function
`context.WithValue(parent, key, val)`	Attaches request-scoped data, retrieved with ctx.Value(key)
`ctx.Done() / ctx.Err()`	Channel that closes on cancellation; reports why

Coding Challenges

CHALLENGE 1

Write a function countTo(ctx context.Context, n int) that counts from 1 to n, printing each number with a 300ms pause between them, but stops early and prints "Stopped early" if ctx is cancelled. Call it with a context.WithTimeout of 1 second and n = 10, so it stops before reaching 10.

View solution

CHALLENGE 2

Using context.WithCancel, start a goroutine that prints "Working..." every 200ms until cancelled. In main, let it run for about 1 second, then call cancel() and confirm (via a short Sleep and a printed message) that the goroutine stopped.

View solution

CHALLENGE 3

Define a contextKey type and a requestIDKey constant. Use context.WithValue to attach a request ID string to a context, then write a function logRequest(ctx context.Context) that reads it back with ctx.Value and prints it.

View solution

CHAPTER 3 QUICK REFERENCE

- **ctx context.Context** — always the first parameter of a cancellation-aware function
- **context.Background()** — the root context to start from
- **context.WithTimeout(parent, duration)** — auto-cancels after a fixed time; always defer cancel()
- **context.WithCancel(parent)** — manual cancellation via the returned function
- **context.WithValue(parent, key, value)** — attaches request-scoped data
- **ctx.Done()** — a channel that closes when the context is cancelled or times out
- **ctx.Err()** — explains why: context.Canceled or context.DeadlineExceeded
- **Next chapter:** testing — the testing package, table-driven tests, and go test

CHAPTER 4 OF 5

Testing: the testing Package and go test

COURSE 3 · CH 4

Testing: the testing Package, Table-Driven Tests, and go test

Testing is built directly into the language and toolchain — no separate library needs installing to get started

Every chapter so far has tested code by reading `fmt.Println` output by eye — fine for a 5-line example, unworkable for a real program with dozens of functions. Go's standard library includes a `testing` package, and the `go` command includes a built-in test runner, `go test` — together, the standard way Go code gets verified automatically.

The File Naming Convention

```
// math.go
package main

func Add(a, b int) int {
        return a + b
}

// math_test.go
package main

import "testing"

func TestAdd(t *testing.T) {
        result := Add(2, 3)
        if result != 5 {
                t.Errorf("Add(2, 3) = %d; want 5", result)
        }
}
```

A test file must be named `*_test.go` (here, `math_test.go`, testing code in `math.go`) — this naming is how Go's tooling recognises test code and excludes it from a normal build. Each test function's name must start with `Test` and take exactly one parameter, `*testing.T`.

Running Tests with go test

```
# In the terminal:
go test ./...
# ok      example/mathpkg    0.002s

go test -v ./...
# === RUN   TestAdd
# --- PASS: TestAdd (0.00s)
# PASS
# ok      example/mathpkg    0.002s
```

`go test` automatically finds and runs every `Test...` function in `*_test.go` files, with no manual registration needed anywhere. `-v` shows each individual test's name and result, rather than just a single pass/fail summary line.

t.Errorf vs t.Fatalf

```
func TestDivide(t *testing.T) {
        result, err := Divide(10, 0)
        if err == nil {
                t.Fatalf("expected an error, got none")   // stops THIS test immediately
        }
        if result != 0 {
                t.Errorf("expected 0, got %v", result)   // records failure, keeps checking
        }
}
```

`t.Errorf` records a failure but lets the rest of the test function keep running — useful when several independent checks are worth reporting together. `t.Fatalf` records a failure AND stops that test function immediately, appropriate when a later check would be meaningless or panic without the earlier one passing first (here, examining `result` wouldn't make sense if `err` turned out to be unexpectedly nil from a successful division).

Table-Driven Tests — The Idiomatic Go Pattern

```
func TestAdd(t *testing.T) {
        tests := []struct {
                a, b, want int
        }{
                {2, 3, 5},
                {-1, 1, 0},
                {0, 0, 0},
        }

        for _, tt := range tests {
                result := Add(tt.a, tt.b)
                if result != tt.want {
                        t.Errorf("Add(%d, %d) = %d; want %d", tt.a, tt.b, result, tt.want)
                }
        }
}
```

Rather than writing a separate test function per case, a **table-driven test** declares a slice of anonymous structs (Intermediate Chapter 1's struct knowledge, combined with Fundamentals Chapter 6's `range`) — each one a distinct input/expected-output pair — and loops over them with one shared assertion. Adding a new test case becomes a one-line addition to the table, rather than an entirely new function.

Subtests with t.Run

```
for _, tt := range tests {
        t.Run(fmt.Sprintf("%d+%d", tt.a, tt.b), func(t *testing.T) {
                result := Add(tt.a, tt.b)
                if result != tt.want {
                        t.Errorf("got %d; want %d", result, tt.want)
                }
        })
}
```

`t.Run("name", func(t *testing.T) { ... })` wraps each table entry as its own named subtest — `go test -v` then reports each case individually (`TestAdd/2+3`, `TestAdd/-1+1`, ...) instead of one combined pass/fail for the whole table, making it immediately clear which specific input failed.

> **GO TEST -RUN**
>
> `go test -run TestAdd` runs only tests whose name matches the given pattern — useful for focusing on one failing test without re-running an entire, possibly slow, test suite.

CONCEPT	CONVENTION
Test file name	`name_test.go`
Test function name	`func TestXxx(t *testing.T)`
Record failure, continue	`t.Errorf(...)`
Record failure, stop this test	`t.Fatalf(...)`
Run all tests	`go test ./...`
Verbose output	`go test -v ./...`

Coding Challenges

CHALLENGE 1

Write a function IsEven(n int) bool, then write a test file with TestIsEven covering at least 3 cases (an even number, an odd number, and zero) using t.Errorf for any mismatch.

View solution

CHALLENGE 2

Rewrite Challenge 1's test as a table-driven test: a slice of struct { input int; want bool } with at least 5 cases, looped over with a single shared assertion.

View solution

CHALLENGE 3

Write a function SafeDivide(a, b float64) (float64, error) returning an error for division by zero. Write a table-driven test using t.Run for named subtests, covering a normal division and a divide-by-zero case, checking both the result/error appropriately for each.

View solution

CHAPTER 4 QUICK REFERENCE

- **name_test.go** — required file naming for Go to recognise test code
- **func TestXxx(t *testing.T)** — required test function signature
- **t.Errorf(...)** — records a failure, test function keeps running

- **t.Fatalf(...)** — records a failure, stops the test function immediately
- **Table-driven tests** — a slice of struct cases looped with one shared assertion
- **t.Run("name", func(t *testing.T) {...})** — named subtests, reported individually
- **go test ./...** / **go test -v ./...** — run all tests, verbosely
- **This completes Go Advanced (Course 3) as currently planned.** A possible Chapter 5 would cover JSON and encoding/json for working with APIs.

CHAPTER 5 OF 5

JSON and encoding/json

COURSE 3 · CH 5

JSON and encoding/json

Converting between Go structs and JSON text — the format almost every real API speaks

Fundamentals Chapter 10 used `response.json()` in JavaScript without much ceremony — JSON parses directly into a plain object there, since JavaScript objects and JSON already share the same shape. Go's static typing (Fundamentals Chapter 2) means converting to and from JSON needs an explicit step: **marshaling** (struct → JSON) and **unmarshaling** (JSON → struct), both handled by the standard library's `encoding/json` package.

Marshaling — Struct to JSON

```
import "encoding/json"

type Person struct {
        Name string
        Age  int
}

p := Person{Name: "Philip", Age: 35}

data, err := json.Marshal(p)
if err != nil {
        fmt.Println("Error:", err)
}

fmt.Println(string(data))   // {"Name":"Philip","Age":35}
```

`json.Marshal` returns the JSON as a `[]byte` slice, plus an `error` — the same Fundamentals Chapter 5 pattern used throughout the language. `string(data)` converts those raw bytes into a printable string. Note that `Name` and `Age` appear capitalised in the JSON output by default — only capitalised (exported) struct fields are visible to `encoding/json` at all.

Struct Tags — Controlling the JSON Output

```go
type Person struct {
        Name string `json:"name"`
        Age  int    `json:"age"`
        Email string `json:"email,omitempty"`
}

p := Person{Name: "Philip", Age: 35}
data, _ := json.Marshal(p)
fmt.Println(string(data))   // {"name":"Philip","age":35} — Email omitted, it's empty
```

The backtick-delimited text after each field is a **struct tag** — metadata read by `encoding/json` at runtime. `json:"name"` renames the field for JSON purposes (lowercase, matching typical API conventions); `omitempty` drops the field entirely from the output when it holds its zero value (Fundamentals Chapter 2) — here, `Email` disappears since it was never set.

> **_ IN THE DISCARDED ERROR**
>
> `data, _ := json.Marshal(p)` uses the blank identifier (Fundamentals Chapter 4) to explicitly throw away the error return value — acceptable in a quick example, but real code should always check it properly, the same way `err != nil` is checked everywhere else in Go.

Unmarshaling — JSON to Struct

```go
jsonData := []byte(`{"name":"Sam","age":28}`)

var p Person
err := json.Unmarshal(jsonData, &p)
if err != nil {
        fmt.Println("Error:", err)
}

fmt.Println(p.Name)   // "Sam"
```

`json.Unmarshal` goes the other direction: raw JSON bytes into a struct. The destination is passed as a **pointer** (`&p`) — exactly Intermediate Chapter 1's pointer-receiver lesson applied here: without the `&`, `Unmarshal` would only be able to fill in a throwaway copy, and `p` back in the caller would remain empty.

> **FORGETTING & BEFORE THE DESTINATION IS A SILENT BUG**
>
> `json.Unmarshal(jsonData, p)` (no `&`) compiles and runs without panicking in many cases, but `p` never actually gets populated — Unmarshal needs a pointer specifically so it can write through to the real variable, the same plain-vs-pointer-receiver distinction from Intermediate Chapter 1.

Fetching JSON from a Real API

```
type Post struct {
        ID    int    `json:"id"`
        Title string `json:"title"`
}

func fetchPost(id int) (Post, error) {
        var post Post

        resp, err := http.Get(fmt.Sprintf("https://jsonplaceholder.typicode.com/posts/%d", id))
        if err != nil {
                return post, err
        }
        defer resp.Body.Close()

        body, err := io.ReadAll(resp.Body)
        if err != nil {
                return post, err
        }

        err = json.Unmarshal(body, &post)
        return post, err
}
```

This is the complete realistic pattern: `http.Get` fetches the response, `io.ReadAll` reads its full body into bytes, `json.Unmarshal` parses those bytes into a struct. `defer resp.Body.Close()` (Intermediate Chapter 3's `defer`) guarantees the response body is closed once the function returns, success or failure — the direct Go equivalent of JavaScript Fundamentals Chapter 10's `fetch` + `response.json()` pair.

JAVASCRIPT	GO
`JSON.stringify(obj)`	`json.Marshal(value)`
`JSON.parse(text)`	`json.Unmarshal(bytes, &target)`
`No equivalent — keys match property names`	`` `json:"key"` struct tag controls the JSON key name ``
`await response.json()`	`io.ReadAll(resp.Body) + json.Unmarshal(body, &target)`

Coding Challenges

CHALLENGE 1

Define a struct Product with fields Name (string), Price (float64), and InStock (bool), with json tags using lowercase key names. Create one, marshal it with json.Marshal, and print the resulting JSON string.

View solution

CHALLENGE 2

Given a raw JSON string `{"name":"Laptop","price":999.99,"inStock":true}`, define a matching struct with appropriate json tags, unmarshal it, and print each field.

View solution

CHALLENGE 3

Write a function fetchUser(id int) (User, error) that fetches from https://jsonplaceholder.typicode.com/users/{id}, unmarshals into a User struct with at least Name and Email fields (with json tags), properly closing the response body with defer. Call it and print the result, handling any error.

View solution

CHAPTER 5 QUICK REFERENCE

- **json.Marshal(value)** — struct/value → JSON []byte, plus an error
- **json.Unmarshal(bytes, &target)** — JSON []byte → struct; target MUST be a pointer
- **Only capitalised (exported) struct fields** are visible to encoding/json
- **`json:"key"`** — struct tag controlling the JSON field name
- **`json:"key,omitempty"`** — omits the field from output if it's the zero value
- **http.Get(url) + io.ReadAll(resp.Body) + json.Unmarshal** — the full fetch-and-parse pattern
- **defer resp.Body.Close()** — always close a response body once done with it
- **This completes Go Advanced (Course 3) and the Go course overall** as currently planned across all 3 courses.